INRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports Techniques

N° 31

LE LANGAGE HELL DE DESCRIPTION DE CIRCUITS INTEGRES

Bertrand SERLET

Décembre 1983

LE LANGAGE HELL DE DESCRIPTION DE CIRCUITS INTEGRES

Bertrand Serlet

I.N.R.I.A.
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay Cedex

RESUME

Nous présentons dans ce rapport le langage "HELL" de description structurelle de circuits, intégré au sein du système de Conception Assistée par Ordinateur pour Circuits Intégrés développé à l'INRIA. Le langage peut servir de base de données unique, et permet une conception hiérarchique. Un simulateur multi-modes opérant à partir d'une description dans ce formalisme sera décrit dans une publication ultérieure.

Mots clefs:

Circuit Intégré, VLSI, CAO, Langage de description de circuits.

ABSTRACT

We present in this report a Hardware Description Language, called "HELL", which is part of the VLSI CAD system developped at INRIA. A main point of the language is the ability to be used as a unique data base for all processors of the system, and to allow a hierarchical conception. A multi-modes simulator operating from a HELL description will be presented in a next publication.

Keywords:

VLSI, CAD, Hardware Description Language.

Imprimé le Vendredi 25 Novembre 1983

SOMMAIRE

1 Nécessité d'un langage de description de circuits intégrés

2 Le langage de programmation 'CEYX'
- 2.1 Le langage d'implémentation LE LISP
- 2.2 Types
- 2.3 Messages et sémantiques
- 2.4 Le type arbre
- 2.5 Conclusion

3 Description du langage HELL
- 3.1 Description du micro-langage
- 3.2 Description du macro-langage
- 3.3 Syntaxe B.N.F. du langage
- 3.4 Problèmes syntaxiques

4 Description de circuits en HELL

5 Conclusion

LE LANGAGE HELL

1 Nécessité d'un langage de description de circuits intégrés

L'apparition récente des VLSI explique le manque d'outils de CAO cohérents, unifiés et complets. L'une des briques de base d'un tel système est un langage de description des masques. C'est dans ce but qu'a été créé le langage LUCIFER [LUCIFER83], dont l'originalité consiste surtout dans l'utilisation intensive de la hiérarchie, et dans le fait que c'est un sur-ensemble d'un langage pré-existant. Autour de cette composante du système de CAO - essentielle puisque sans elle il est impossible de communiquer à l'entreprise le circuit qui doit être réalisé - s'articulent plusieurs outils: éditeur graphique [Lévy83], vérificateur de règles de dessin [Gallot83], extracteur [Heintz82].

Mais une autre pièce indispensable au système est un langage de description de haut niveau de circuits et d'algorithmes [Moto-oka83]. C'est, par exemple, une description dans ce formalisme, donnée en sortie de l'extracteur, que les simulateurs - outils indispensables à la conception de circuits complexes - doivent accepter en entrée. Enfin, et surtout, le concepteur de circuits, avant de se lancer dans le dessin des masques, spécifie le fonctionnement du circuit, et c'est cette spécification fonctionnelle qui dirige le dessin des masques, même si le détail du dessin des masques peut avoir des répercussions importantes sur la spécification de haut niveau.

C'est pour répondre a de tels besoins qu'a été conçu le langage HELL. HELL est un langage de description hiérarchique de connexions de modules. A l'instar de son aîné LUCIFER, HELL est basé sur CEYX [Hullot83], lui-même sur-ensemble de LE_LISP [Chailloux83a], et nous commencerons donc par présenter CEYX.

Nous spécifierons ensuite le langage HELL, sa syntaxe, et sa sémantique en temps que langage de description structurel.

Enfin nous indiquerons brièvement les liens entre ce langage et les processeurs autres que les simulateurs.

2 Le langage de programmation 'CEYX'

CEYX est un environnement de programmation fournissant à l'utilisateur des mécanismes de haut niveau pour construire des objets, et définir des algorithmes sur ces objets. La plupart des notions introduites par CEYX se retrouvent dans divers langages de programmation. Ainsi, les mécanismes de typage doivent beaucoup aux langages typés, comme Algol, Pascal ou ML. Les mécanismes de messages se sont inspirés de ceux de Simula ou de Smalltalk [Smalltalk82]. Mais l'originalité de CEYX est à la fois de fournir des moyens cohérents, simples et efficaces d'utiliser ces notions, et de s'appuyer sur un langage presque dépourvu de structures de données. Ce langage est LISP, ou plutôt un dialecte de LISP: "LE_LISP".

Ce dialecte présente la particularité de disposer de structures de contrôle évoluées, d'être rapide tant interprété que compilé, et d'être très facilement portable.

Ce chapître ne prétend pas décrire en détail les systèmes LE_LISP [Chailloux83a] et CEYX [Hullot83], mais seulement donner un aperçu des possibilités offertes, utilisées aussi bien dans l'implémentation du langage HELL et de ses processeurs que dans sa définition elle même, comme nous le verrons pour les macro-constructeurs.

Nous allons donc présenter successivement LE_LISP, les types selon la vision CEYX, les mécanismes de messages, et nous finirons par donner un exemple, fondamental pour l'implémentation de HELL.

2.1 Le langage d'implémentation LE_LISP

2.1.1 Historique

LISP compte avec Fortran parmi l'un des plus vieux langages encore utilisés. Créé dans les années 60 [McCarthy62], il a surtout été employé dans la recherche et l'enseignement, mais n'a que rarement pénétré l'industrie, ne serait ce qu'à cause de son auréole de formalisation, le lambda calcul, et de sa faiblesse dans les structures de données et de contrôle, dûe au souci de minimalité de ses créateurs.

Rajeuni au MIT dans le milieu des années 70 par l'apparition du dialecte Maclisp, offrant une plus large bibliothèque de fonctions et quelques types de données, il a servi à la construction de deux gros programmes: l'éditeur Emacs [Greenberg79], précurseur des éditeurs plein écran, et le système de calcul symbolique Macsyma. Le souci d'un faire un petit système portable sur un ordinateur individuel a conduit à l'apparition de LE_LISP [Chailloux82], proche de Maclisp, mais enrichi par l'apport d'algorithmes efficaces pour son interprétation [Chailloux80], et bénéficiant de l'expérience du dialecte Vlisp [Greussay77, Chailloux80].

2.1.2 Structures de contrôle

Des leur création, Fortran et LISP se sont differenciés par leur façon d'exécuter plusieurs fois les mêmes instructions. En Fortran, ce fut l'itération, en LISP, la récursion. Or, il est apparu par la suite que ces deux mécanismes, bien qu'équivalents en théorie, ne le sont pas sur le plan de la facilité d'écriture. Les deux exemples les plus simples en sont le parcours d'un arbre, typiquement récursif, et qui s'écrit trivialement en LISP, et l'itération d'un indice à travers un intervalle, usage typique du "do" de Fortran. Un langage de programmation raisonnablement universel se doit donc de proposer les deux techniques. LE_LISP propose donc les structures de contrôle disponibles dans tous les langages récents: if, when, unless, selectq, while, until, repeat, for, ...

Dans certaines applications, il est aussi nécessaire d'avoir des dispositifs d'échappement pour les exceptions. LE_LISP là encore propose un mécanisme général et unique de contrôle dynamique, et la construction de systèmes s'en trouve donc grandement facilitée.

2.1.3 Implémentation et portabilité

Une originalité du système LE_LISP est d'avoir dégagé une machine virtuelle [Chailloux83b], appelée "LLM3", connaissant certaines opérations de base sur les listes, comme les opérations "CAR" et "CDR". C'est dans le langage de cette machine qu'est écrit l'interpréteur LE_LISP, c'est à dire la fonction "eval", et certaines fonctions usuelles, critiques en temps. C'est aussi ce format que produit le compilateur, qui accélère donc l'interprétation d'une fonction en descendant d'un niveau d'abstraction sa description. Les instructions LLM3 sont macro-générées dans le langage réel d'une machine cible. Pour aller plus vite encore, on peut microcoder ces instructions sur la machine cible, et si cela reste insuffisant, il suffit de descendre encore un niveau d'abstraction en codant les algorithmes sur silicium.

Cette approche permet de diviser la complexité d'écriture des fonctions et du compilateur, et cela se traduit, d'une façon surprenante, par une plus grande vitesse du système.

Par ailleurs, pour transporter le système sur une machine réelle, il suffit de définir les macros LLM3, et les nombreux transports ont prouvé la simplicité de la manœuvre.

2.1.4 Structures de données

C'est dans les structures de données que subsiste une grande faiblesse de LISP. Mise à part la structure de liste, pour laquelle de nombreuses fonctions sont disponibles, LISP ne propose aucun mécanisme de définition et d'utilisation de données complexes. C'est ce problème que CEYX s'efforce de résoudre.

Cependant cette structure de liste est très générale, et surtout permet de traiter de façon symétrique programmes et données. Cela se traduit par une très grande facilité d'extension du langage lui même, c'est à dire de réalisation de sur-systèmes. C'est sans doute cette raison qui a maintenu à travers les années l'utilisation de LISP dans des domaines comme l'intelligence artificielle.

Un autre avantage de la programmation fonctionnelle (effectuée par exemple avec des macros) est d'éviter au programmeur d'écrire deux fois un programme similaire: si deux morceaux de code se ressemblent, c'est dans la plupart des situations qu'ils sont des cas particuliers d'un algorithme fonctionnel plus général.

2.2 Types

La notion de type est souvent présente dans les sciences. En physique, elle se concrétise par la notion d'unité, en mathématique par celle d'ensemble, en biologie par celle d'espèce ...

L'expérience des langages de programmation a prouvé que l'utilisation de types en informatique est d'une grande aide pour l'écriture d'algorithmes, leur mise au point, et leur lisibilité. L'expérience des langages "orientés objets" montre que l'utilisation de types permet aussi d'obtenir des mécanismes simples d'héritage. Il est donc nécessaire de disposer de cette possibilité dans tout langage de programmation évolué, et nous allons maintenant décrire comment CEYX introduit le typage en LISP.

2.2.1 Types simples

Les types simples de CEYX comprennent tous les types atomiques de LISP (nombres, booléens, symboles, chaînes de caractères). En outre, il est possible de créer de nouveaux types simples en définissant une fonction de discrimination qui utilise les fonctions de discrimination des types atomiques. Ainsi, on peut définir l'ensemble des nombres positifs comme étant l'ensemble de tous les nombres pour lesquels le prédicat LISP ">=0" retourne "vrai".
Cela permet aussi de cacher l'implémentation d'un type complexe en en faisant un type atomique, c'est à dire une abstraction du type complexe.

2.2.2 Les types composés list et array

Lorsque l'on réunit plusieurs objets d'un même type "type" on forme, selon un point de vue mathématique, un ensemble de ces objets. Si, en outre, ces objets sont rangés, on obtient une suite.
Le point de vue informatique consiste à considérer un ensemble d'objets de type <T> comme un seul objet de type: "(list <T>)" - en employant la notation utilisée dans CEYX - . L'opérateur "list" est un constructeur de types composés, c'est à dire une fonctionnelle sur l'ensemble des types.

En LISP, une excellente implémentation de la liste CEYX est la liste LISP, vu le nombre de fonctions de manipulations déjà disponibles. Cependant la liste CEYX est moins générale: tous les objets de la liste doivent être du même type.

Un constructeur de type tout à fait similaire à "list" est "array": la distinction tient à la représentation interne, à l'extensibilité, et aux opérations primitives de création et d'accès. Les itérations se font facilement tant sur les listes que sur les tableaux, mais la récursion paraît plus naturelle sur une liste.

2.2.3 Le type composé record

Il est souvent impératif pour programmer un algorithme, de regrouper plusieurs objets de types différents en un même objet. La notion mathématique correspondante est celle de produit cartésien. Pour ce faire, on dispose dans CEYX d'un constructeur de type "record" (enregistrement en français), qui permet ce produit cartésien. On indique en outre, en définissant une instance de type enregistrement, une stratégie de nommage pour accéder à chacune des composantes.
Les fonctions d'accès en lecture et en écriture d'un enregistrement sont implémentées sous la forme de macros et non de fonctions, ce qui permet d'obtenir une rapidité maximale à l'exécution.

2.2.4 Association de noms aux types

Pour désigner un type, il est plus facile d'utiliser un nom. Cette association d'un nom à une donnée, ici un type, est d'ailleurs un mécanisme présent dans tous les langages, l'affectation. Mais on notera que, pour des raisons de lisibilité, on possède plusieurs affectations, une pour chaque type de donnée.

Ainsi sous CEYX, "defrecord" associe un type composé du genre enregistrement à un nom.

Par exemple,

```
(defrecord personne
        prénom ~ string   marié ~ boolean)
```

crée un type enregistrement, produit cartésien du type "string" et du type "boolean", et associe le nom "personne" à ce type. Le macro-caractère "~" sert en CEYX à préciser un type. Un effet de bord de la fonction defrecord, est de créer des fonctions d'accès aux composantes de noms "prénom" et "marié".

Pour les types nommés, CEYX autorise un nouveau mécanisme: l'héritage des propriétés sémantiques, que nous verrons plus loin.

2.2.5 Typage des objets

Un objet représente toute réunion de bits susceptible de contenir de l'information. A tout objet, on peut associer la façon dont l'information y est contenue, c'est à dire le schéma d'utilisation des bits, ou encore le type.

Le type d'un objet est enregistré dans l'objet lui même, afin que les processeurs puissent décider, suivant les données qu'ils reçoivent, du traitement à effectuer.

Cela oblige à perdre un peu en mémoire, puisque de nombreux processeurs ne sont susceptibles que de recevoir des données d'un certain type. Malgré cela, c'est, semble-t-il, l'orientation actuelle de différents langages, et même de certaines architectures de machines.

2.3 Messages et sémantiques

Une fois définies les notions de type et d'objets typés, nous allons leur associer des notions d'actions. On précise donc pour un type particulier d'objet, le comportement d'une instance de cet objet vis à vis d'une certaine action. Nous appelons sémantique ce comportement, et message le lancement de l'action. Dans la terminologie Smalltalk, il est question de méthodes au lieu de sémantiques, et de classes et non de types [Smalltalk82].

Pour définir sous CEYX la sémantique d'un objet vis à vis d'un message, il existe la fonction "defsem", dont la syntaxe ressemble à la fonction LISP de définition de fonctions "de".

Exemple:

```
(defsem (integer multiplie_par_2) (n) (+ n n))
```

2.3.1 Exécution d'une sémantique

Mais il ne suffit pas de définir des sémantiques, il faut pouvoir lancer des messages aux objets. On peut là faire le parallèle avec l'évaluation d'une fonction précédemment définie: définir une fonction, c'est accrocher une certaine sémantique à un symbole, et évaluer une fonction, c'est lancer le message "eval" à ce symbole.

Outre la passation de l'objet - et donc du nom de la fonction à exécuter par l'intermédiaire du type de l'objet - on peut passer à l'évaluateur d'autres arguments. La fonction de lancement de message s'appelle "send".

Exemple:

```
(defsem (integer factorielle) (n)
        (if (= n 0)
            1                                    ; si n = 0
            (* n (send 'factorielle (1- n))))) ; sinon
```

2.3.2 Surcharge d'opérateurs

Grâce au fait que tout objet connaît son type, il est possible de donner une sémantique vis à vis du même message à des symboles différents. En effet, l'objet qui reçoit le message connait son type, et peut donc appliquer la sémantique appropriée.

Par exemple, on peut définir une addition pour les entiers, une addition pour les réels, une addition pour les complexes, et une addition de matrices. Désigner ces 4 opérations par le même symbole s'appelle "surcharge" du symbole (overloading en anglais). Cela permet de simplifier l'écriture des programmes et les rend polymorphes, c'est à dire applicables à toute une catégorie d'objets.

Bien que ce ne soit pas le cas actuellement en CEYX, on peut dans la majeure partie des utilisations déterminer de façon statique, à la compilation, la sémantique à utiliser.

On obtient ainsi à moindre coût les mécanismes de surcharge disponibles en Ada [Ada83].

2.3.3 Dépendance hiérarchique des sémantiques

Sans aucun autre mécanisme, le lancement de sémantiques n'apporte rien de fondamentalement nouveau par rapport à l'exécution traditionnelle de fonctions, et donne seulement une façon différente d'exprimer des algorithmes, peut être plus naturelle pour certains d'entre eux.

En revanche, le mécanisme d'héritage des propriétés sémantiques, va amener une richesse nouvelle au typage des objets. L'idée de base est la suivante: les types forment une hiérarchie, une structure de graphe orienté sans cycle, et, pour un message donné, on hérite de la sémantique des types pères.

Notons qu'il existe un constructeur de type spécial, apparenté aux "flavors" de la "Machine_LISP", permettant d'hériter de plusieurs types différents, et facilitant l'incrémentalité du système de types, en autorisant, conceptuellement, l'extension de types déjà définis. Cette façon d'hériter s'appelle héritage multiple.

2.3.4 Sémantiques ou procédures ?

Il a été démontré très tôt que l'on peut faire n'importe quoi dans n'importe quel langage aussi évolué que celui d'une machine de Turing. Par choix, CEYX est un sur-ensemble de nombreux langages existants. Cependant c'est un sur ensemble cohérent, dans la mesure où tous les mécanismes se trouvent exécutés par la machine virtuelle LE_LISP, machine de très bas niveau.

Toute définition de sémantique va donc se trouver "câblée" par une définition de procédure. En Smalltalk, c'est le contraire qui a été fait, et les procédures sont des objets répondant au message "eval".

D'après notre expérience de programmation, il semble que les processeurs gérant des bases de données (typiquement un éditeur) s'écrivent de façon très esthétique avec les sémantiques, alors que pour beaucoup de calculs algorithmiques, l'approche par les sémantiques ne représente qu'une perte d'efficacité.

Cette perte d'efficacité sera sans doute de moins en moins importante grâce aux ULSI de demain, et il faudra peut-être recâbler la machine CEYX ... !
Dans le futur, on peut aussi espérer que des compilateurs évolués pourront tirer profit de situations typiquement procédurales pour générer un code aussi efficace que dans des langages classiques.

2.4 Le type arbre

Dans tous les domaines, une structuration qui s'est révélée particulièrement efficace est celle d'arbre. Ce type est à lui seul une méthodologie: le découpage d'une tâche importante en sous-tâches plus simples...

2.4.1 Description

Un arbre est composé de nœuds, qui peuvent être de genres différents, et de feuilles. Une feuille n'est qu'un nœud sans descendance. Les informations contenues dans un nœud sont donc:
- le type du nœud
- la liste des fils du nœud, comprenant éventuellement les paramètres du nœud.

La définition d'un arbre ressemble donc simplement à:

```
(defrecord tree  sons)
```

où sons contient la liste des fils et des paramètres.

Les divers nœuds d'un arbre se différencient par le type du nœud, et par la valeur de leur champ sons.

2.4.2 Univers

La construction "defuniverse" permet de préciser des arbres plus particuliers, des familles d'arbres dans lesquelles chaque arbre comporte les mêmes champs supplémentaires.

Exemple:

```
(defuniverse arbre-abc    a  b  c)
```

Toutes les instances d'arbre de type "arbre-abc" ont des champs a, b, et c

Conceptuellement, "(defuniverse arbre-abc a b c)" est équivalent à "(defrecord arbre-abc sons a b c)". Tous les "univers" ont le même champ "sons", ce qui permet l'écriture de processeurs généraux opérant sur ces arbres.

En outre "defuniverse" permet de donner une filiation, de définir l'héritage en

précisant l'univers père. Par défaut, l'univers père est l'univers de tous les arbres, propriété particulièrement utilisée dans l'éditeur de structures BIGMACS [Hullot84].

Exemple:

```
(defuniverse arbre-abcd ~ arbre-abc d)
```

Toutes les instances d'arbre de type "arbre-abcd" ont 4 champs a, b, c, et d, et héritent des propriétés des arbres de type "arbre-abc"

2.4.3 Constructeurs

La construction "defcons" permet de définir simplement une fonction de création d'objets d'un certain type, type fils d'un type d'univers. En général, on définira un constructeur pour chaque type de nœud, et c'est sur le constructeur que l'on définira des propriétés sémantiques.

Ainsi,

```
(defcons divide ~ expressions (integer integer))
```

crée une fonction "divide" à 2 arguments entiers, qui retourne un objet de type divide, et dont les héritages sémantiques sont ceux des types: divide, expressions, tree dans cet ordre.

2.4.4 Macro-constructeurs

Un macro-constructeur est essentiellement un constructeur, pour lequel on inclut dans sa définition la sémantique du type correspondant vis à vis du message "expand".
Lors de l'envoi du message M à un certain objet, le message expand est par convention celui que l'on applique tant que l'objet ne répond pas au message M.

Cela permet d'augmenter aisément le langage, c'est à dire l'ensemble des types d'arbres de l'univers considéré, sans pour autant compliquer les processeurs, car, en règle générale, on ne donne pas de sémantique pour les macro-constructeurs.

Ainsi, l'écriture d'un processeur se ramène à la définition de sémantiques pour les constructeurs.

Notons enfin que le nom est dû au fait qu'il s'agit là d'un mécanisme similaire à la macro-expansion en LISP.

2.5 Conclusion

LE_LISP allie donc à la simplicité et à la généralité de LISP, la puissance apportée par des structures de contrôle developpées, une vitesse tout à fait comparable à des langages de types Algol, et une très grande portabilité.
La possibilité de décrire et utiliser des structures complexes est apportée par CEYX.
La réunion de ces deux systèmes forme donc un environnement de programmation particulièrement agréable pour l'écriture de systèmes

importants, d'une manière concise, fiable, et efficace.

3 Description du langage HELL

Comme il l'a été annoncé précédemment, LISP est à la fois langage d'implémentation du système et langage de commande, et donc, pour commodité, le langage HELL que nous allons décrire emprunte aussi à LISP sa syntaxe.
Certains passages de la suite de ce chapître seront donc beaucoup plus facilement compréhensibles aux habitués de la syntaxe LISP. Cependant une ambition du langage est de permettre, moyennant une bibliothèque d'utilitaires, une écriture aisée de circuits, même complexes, par un néophyte en LISP.

Beaucoup de choix, comme la façon de passer les paramètres, ont été fait en étudiant la solution proposée par les langages de type Pascal (Ada, Algol, ...). Nous nous efforcerons donc de mettre en relief, au fur et à mesure de l'introduction de HELL, les différences et les similitudes avec ces langages, que nous désignerons sous le nom générique de "langages pascaliens". Nous essaierons aussi d'indiquer les choix différents faits dans d'autres langages de description de circuits, et les motivations de ces choix.

Nous appelerons micro-langage l'ensemble des mécanismes indispensables pour décrire des circuits, et macro-langage le langage complet, incluant toutes les facilités auxquels le concepteur de circuits complexes est en droit de s'attendre, mais indépendammant de toute technologie ou de toute bibliothèque.

3.1 Description du micro-langage

3.1.1 En tête d'un module

Dans toute la suite, nous emploierons le mot "module" pour désigner une boîte noire d'où sortent quelques canaux de communication avec le monde extérieur, et qui lui permettent de remplir sa fonction. Nous allons nous spécialiser dans la description de circuits VLSI, et donc nous utiliserons le mot "fil" pour désigner tout canal de communication.

Dans la plupart des langages de programmation, le nom des paramètres d'une fonction importe peu pour l'extérieur de cette fonction, par exemple pour son appel, et seul l'ordre des paramètres compte. De façon similaire nous avons choisi de numéroter les fils qui dépassent d'une boîte noire, plutôt que de les nommer.

Comme pour les langages pascaliens, les fils sont typés, et ce de deux façons. D'abord par une orientation de l'échange d'information qui se fera dans ce fil. Ensuite par une structuration des fils eux mêmes. Nous reviendrons sur ce typage et cette structuration, qui se retrouvent dans les langages pascaliens.
Nous appelerons ports d'entrée sortie l'ensemble des fils typés qui dépassent

de la boîte noire.
Nous avons donc défini l'entête d'un module, c'est à dire la seule information disponible quand on ne regarde pas l'intérieur du module.

3.1.2 Connection de plusieurs modules

Une fois définies les boîtes noires, il faut préciser comment nous allons composer plusieurs petites boîtes noires pour en obtenir une grosse ...
De cette façon, nous allons pouvoir hiérarchiser une description de circuit, méthodologie universellement reconnue comme étant indispensable dans tout langage, tant de programmation que de description de circuits.

La communication entre plusieurs modules se fait par l'intermédiaire de fils, de même qu'elle se fait par l'intermédiaire de variables dans un langage de programmation universel. Une connexion de plusieurs modules est donc formée de fils nommés et d'instances de sous modules, ces derniers ayant leurs ports d'entrée sortie nommés.

L'opérateur de connexion s'appelle "connect". L'opérateur d'instanciation et de nommage s'appelle "block". La déclaration des fils se fait avec "wires", et on précise les entrées et les sorties par "in" ou "out".

Exemple du NOT en technologie Nmos:

Dans la technologie Nmos, on distingue deux types de transistors, que nous appelons "trn" (transistor normal) et "trd" (transistor à déplétion servant à former des résistances "pull-up"). Nous allons supposer que le langage connaît ces deux types de transistors, point sur lequel nous reviendrons en détail plus loin.

En technologie Nmos un NOT - porte inversant une entrée arbitrairement nommée "e", utilisant l'alimentation "vdd" et la masse "gnd", et ayant une sortie "s" - est formée de 2 transistors:
- un pullup, transistor à déplétion "trd" dont la grille et la source sont connectés à "s", et dont le drain est relié à la source de tension "vdd".
- un transistor normal "trn" pour lequel la grille est l'entrée "e" de la porte, la source la masse "gnd", et le drain la sortie "s".

La description d'une telle porte en HELL sera:

```
(connect                    ; l'opérateur de composition
   (wires                   ; déclaration des fils du module
      in  (e gnd vdd)       ; les ports d'entrée
      out (s))              ; les ports de sortie
   (block                   ; l'opérateur d'instanciation et de nommage
     trd                    ; un transistor à déplétion
       s s vdd)             ; les paramètres dans l'ordre grille-source-drain
   (block
     trn                    ; un transistor normal
       e gnd s))
```

3.1.3 Fils locaux à un module

Dans l'exemple précédent tous les fils utilisés étaient ports d'entrée sortie du module composé.
Lorsque ce n'est pas le cas, on précise par le mot clef "intern" utilisé à la place de "in" ou "out" que le fil est interne au module, invisible de l'extérieur.

Cela donne pour le "NAND":

```
(connect
    (wires
       in (entree1 entree2 gnd vdd)
       out (sortie-du-NAND)
       intern (fil-interne))      ; le fil local au module
    (block   trd   sortie-du-NAND sortie-du-NAND vdd)
    (block   trn   entree1 sortie-du-NAND fil-interne)
    (block   trn   entree2 fil-interne gnd))
```

Bien sûr, une fois de plus, on peut faire la comparaison avec un langage universel de programmation déclaratif, où, comme pour HELL, toute variable (ici, tout fil) doit être déclarée avant d'être utilisée.

3.1.4 Définition de module

Jusque là nous avons décrit des modules, c'est à dire des objets définis par (connect ...), mais nous n'avions aucun mécanisme permettant d'attacher un objet HELL à un nom afin de pouvoir le réutiliser par la suite. Ce mécanisme, cas particulier d'affectation, est introduit par la fonction à effet de bord "defblock", qui ressemble sur le plan syntaxique à connect, mais pour laquelle on précise un nom:

```
(defblock NOT
    (wires
        in  (e gnd vdd)
        out (s))
    (block trd s s vdd)
    (block trn e gnd s))
```

Désormais, on peut utiliser le symbole NOT à la place de (connect ...). NOT obtient donc le statut de "symbole HELL": tous les processeurs en rencontrant un symbole HELL, c'est à dire un symbole auquel est attaché une certaine sémantique HELL, remplacent le symbole par sa sémantique. Nous appelons "nœud HELL" une expression LISP qui est soit un symbole HELL, soit un objet composé obtenu par (connect ...), et qui représente un module.

Notons que dans une instanciation par (block ...), l'expression suivant block est un nœud HELL, et pas seulement un symbole HELL. Donc on peut écrire de façon équivalente (block NOT a b) ou (block (connect (wires ...) ...) a b).
Remarquons enfin que rien n'oblige, lorsque l'on écrit (block nom ...) que nom soit le nom d'un symbole HELL déjà défini, ce qui laisse une grande liberté dans l'ordre dans lequel on décrit un circuit.

3.1.5 Instance d'un module

Maintenant que le type "nœud HELL" est défini, nous pouvons préciser la syntaxe et le rôle de l'opérateur "block".

Dans le contexte d'une connexion, c'est à dire sous la portée de l'opérateur connect ou defblock, block permet d'instancier un nœud HELL, en indiquant des noms effectifs de paramètres au module instancié. C'est l'équivalent de l'appel d'une procédure dans un langage pascalien. Nous avons choisi pour HELL d'écrire explicitement qu'il s'agit d'une instanciation principalement pour des raisons syntaxiques de simplicité et d'homogénéité avec LISP.

Ainsi, l'opérateur block permet le renommage externe des ports d'entrée sortie d'un module.

3.1.6 Typage des ports d'entrée sortie

Jusque là, tous les modules que nous avons définis ne possèdent, comme ports de communication, que des entrées ou des sorties. Dans certaines technologies, le mos par exemple, il est souvent nécessaire que des modules aient des ports bidirectionnels. Aux typages par "in" ou "out", nous rajoutons donc le typage "inout".

Notons que divers synonymes de in,out et inout existent, et nous emploierons indifféremment "input" pour in, "output" pour out, et "io" pour inout.

Le même typage directionnel existe dans les langages pascaliens.

La sémantique de ce typage est utilisée par certains processeurs, le simulateur fonctionnel, par exemple, et ces processeurs peuvent demander à ce que certaines contraintes de typage soient satisfaites.

3.1.7 Modules primitifs

Dans les langages pascaliens toutes les fonctions primitives ne peuvent pas se décrire en restant dans le langage. De même, dans la plupart des autres langages de description de circuits, certains modules sont privilégiés, les portes de base, par exemple, mais aussi parfois le point registre [Zeus83].

En revanche, en HELL, les modules primitifs "trn" et "trd" eux mêmes peuvent se décrire avec defblock de la façon suivante:

```
(defblock trn
          (wires
             input (grille)
             io (source drain)))
```

Cette définition "vide" de l'objet trn ne doit pas choquer, on peut décrire de façon similaire un "NAND". La seule information qui compte, pour un processeur donné, c'est la sémantique, vis à vis de ce processeur, que l'on donne aux objets HELL. Par exemple, pour un simulateur de portes logiques, il suffit d'accrocher une certaine sémantique aux portes logiques NAND et NOR, et alors peu importe si ces portes sont ou non des objets vides. Pour un simulateur "switch-level" c'est sur les transistors que l'on accroche de la sémantique. Pour un simulateur analogique, une définition plus appropriée de trn serait du genre:

```
(defblock trn
          (wires
```

```
        input (grille)
        io (source drain))
    (block capa grille)
    (block resistance source drain))
```

D'autres langages de description de circuits, pour éviter de figer les possibilités sémantiques des briques de base dont l'utilisateur dispose, comme Hisdl [Hisdl82], laissent la sémantique des primitives sous le contrôle de l'utilisateur, mais maintiennent une distinction entre cellules composées et cellules primitives.

Outre la complexité plus grande du langage que cela entraîne, cette approche ne permet pas, contrairement à HELL, d'arrêter la description d'un circuit au niveau de précision choisi: on peut décrire aussi bien en HELL des plans de masse - toutefois sans information géometrique - que des portes. De cette façon, il est possible de rendre la description d'un circuit relativement indépendante de la technologie, en choisissant des briques de base de suffisamment haut niveau.

3.2 Description du macro-langage

3.2.1 Paramètres implicites

Il devient très vite fastidieux d'écrire tous les paramètres d'un circuit, plus particulièrement les noms des alimentations comme gnd et vdd, que l'on retrouve dans presque toute définition de symbole HELL.

Le même problème a été rencontré dans la plupart des langages de programmation:
- C'est l'output implicite dans un "write" en Pascal, avec toutes ses ambiguïtés.
- C'est la possibilité de certains LISP d'avoir des arguments optionnels ayant des valeurs par défaut.
- C'est la valeur par défaut des paramètres d'une fonction Ada, avec les restrictions syntaxiques que cela impose.

HELL permet d'obtenir simplement une solution à ce problème, au coût de la notion de portée dynamique: Si "vdd" est un fil du nœud HELL A et si le nœud HELL A se compose du nœud HELL B, la portée dynamique du fil "vdd" s'étend au module B: B peut utiliser "vdd" en précisant que c'est un nom "global" défini plus haut.

En utilisant le mécanisme de noms globaux, le pull-up en Nmos se décrit de la façon suivante:

```
(defblock pu
        (wires
            io (x)
            input global (vdd))
        (block trd x x vdd))
```

De même NAND s'écrit:

```
(defblock NAND
          (wires
              input (entree1 entree2)
              output (sortie-du-NAND)
              input global (gnd)
              intern (fil-interne))
          (block   pu    sortie-du-NAND)
          (block   trn   entree1 sortie-du-NAND fil-interne)
          (block   trn   entree2 fil-interne gnd))
```

Il n'est pas nécessaire de préciser que "vdd" est un nom global pour le NAND: conceptuellement, les processeurs rajoutent, en une première passe sur la description HELL, les noms globaux implicites tels "vdd".

Bien sûr tous les types (in, out, inout et leurs synonymes) peuvent être précisés avant le mot clef "global".

3.2.2 Structuration des fils

L'électronicien ou le concepteur Vlsi ont souvent besoin de traiter plusieurs fils simultanément, de "structurer" les fils eux mêmes. L'exemple le plus répandu de structuration est bien sûr le bus, mais ne se limite pas à cet exemple. Pour des raisons de simplicité, il peut être commode de désigner les signaux "vdd" et "gnd" sous le seul terme "alim". Il est aussi parfois utile, sous certaines technologies, de calculer pour chaque signal le signal complémentaire: bien que ne voulant pas savoir ce détail au niveau de la description fonctionnelle, on veut le connaître au plus bas niveau, tout en n'ayant qu'une seule description.

C'est cela que permet la structuration des fils.

Un fil structuré est déclaré à l'intérieur d'un "wires" par un mot clef comme "bus" ou "list" . L'utilisateur a la possibilité de se définir lui même de nouveaux mots clefs (moyennant l'écriture d'une courte fonction LISP).

Un fil structuré se référence soit par son nom, soit par le nom de l'une de ses composantes.

Nous n'allons considérer que les mots clefs prédéfinis "bus" et "list". La syntaxe de l'argument suivant le mot clef est:
- pour bus : bus (<nom> <min> <max>)
- pour list: list (<nom> (<suite_de_noms>))

Ces deux mots clefs permettent de créer des fils structurés ayant un seul niveau de stucturation.

Exemples:

```
(wires in list (alim (gnd vdd)))
```

crée un fil structuré à 2 composantes de noms gnd et vdd.

```
(wires in list (contrôle (x y z)))
```

crée un fil structuré à 3 composantes.

```
(wires out bus (data 1 8))
```

crée un fil structuré à 8 composantes de noms: data[1], data[2], ..., data[8].

Si data a été defini par "... bus (data 1 8) ..." , et si l'on utilise le fil structuré data sans détailler ses composantes, comme dans "(block alu data)", alors la définition des ports d'entrée sortie du module, ici "alu", devra ressembler à:

```
(wires io bus (données 0 7))
```

ou à

```
(wires out list (d (d1 d2 d3 d4 x1 x2 x3 x4)))
```

Elle devra de toute façon comporter un fil structuré à 8 composantes. L'algorithme utilisé par les processeurs pour vérifier cette correspondance de structuration est celui d'un "match d'arbre".

Il existe une très forte ressemblance entre la stucturation des fils en HELL et la structuration des variables dans un langage pascalien. La structuration induite par bus ressemble à celle d'un tableau Pascal, et list ressemble a' l'enregistrement Pascal, à la stratégie de nommage près.

Notons enfin, qu'il est possible de regrouper plusieurs fils en un fil structuré lors du nommage des ports d'entrée sortie lors d'une instanciation de module. Ainsi, si "XOR2" est le nom d'un module à 2 ports, un d'entrée, structuré, à 2 composantes, et un de sortie, non structuré, et si on veut appliquer XOR2 aux signaux "a" et "b" et ramener la sortie dans le signal "s", on écrira:

```
(block XOR2 (a b) s).
```

Parfois aussi on a besoin de calculer un nom de fil pour le passer en paramètre. Dans ce cas, on fait précèder le nom à calculer du macro-caractère "%". Cet horrible détail syntaxique permet de donner une syntaxe agréable pour manipuler des composantes de bus, grâce aux macro-caractères "[" et "]", qui combinent évaluation et application d'une fonction "indice" qui indice un symbole.

```
(indice 'adresse 4)       ->  adresse[4]
(indice 'donnée (1+ 2))   ->  donnée[2]
```

Après un crochet ouvrant, on écrit le symbole à indicer, suivi de l'indice puis du crochet fermant. Ainsi la cinquième composante du bus B se désignera par [B 5], et la i-ème par [B i].

L'utilisation de "[" et "]" cache celle de "%", et donc "%" est surtout un mécanisme interne d'implémentation.

3.2.3 Immersion dans un langage de programmation

La conception de circuits intégrés est un domaine de technologues, mais surtout d'informaticiens ... Le concepteur informaticien veut utiliser son langage de description de circuits au sein d'un environnement de programmation.

Par exemple, il veut paramétrer ses descriptions de circuits.

Pour ce faire, il a deux solutions:
- développer un environnement de programmation spécifique
- utiliser un environnement de programmation préexistant.

La première solution semble présenter comme seul avantage celui de pouvoir disposer de mécanismes ad hoc, bien adaptés à la description de circuits. En fait, il n'en est rien: l'expérience prouve que les mécanismes les plus généraux sont les plus simples, les plus profonds, les plus faciles à implémenter et aussi les plus efficaces.

Nous avons donc choisi d'immerger HELL dans l'environnement CEYX.

Ainsi, connect, wires, block sont des fonctions LISP retournant des objets d'un type donné (au sens de CEYX), et toute fonction retournant des objets de même type peut être utilisée: l'exemple qui suit est la description d'un circuit arborescent de taille arbitraire.

```
(defblock feuille       ; pour le besoin de l'exemple,
          (wires        ; l'en tête de feuille suffit
            out (x)))

(defblock noeud         ; idem pour nœud
          (wires
             in (x-gauche x-droit)
             out (resultat)))

(de arbre (n)                   ; "de" est la fonction LE_LISP
                        ; de définition de fonctions
    (if (= n 1)         ; le test, en LE_LISP
        'feuille        ; si n=1
        (connect        ; si n<>1
            (wires
              out (resultat)
              intern (x-gauche x-droit))
            (block (arbre (div n 2)) x-gauche)
            (block (arbre (div n 2)) x-droit )
            (block noeud x-gauche x-droit resultat))))
```

Une fois évaluée, l'expression LISP (arbre <n>) est du type nœud HELL, et représente donc un module. Sans introduire quoi que ce soit de nouveau dans le langage, ce qui facilite l'écriture des processeurs travaillant sur une description HELL, nous avons donc décrit une structure relativement complexe. Des vérifications de type sont faites au fur et à mesure de la création de nouveaux objets, et donc on assure ainsi la consistance de la structure de données.

3.2.4 Macro-constructeurs de modules

Malheureusement, un reproche que l'on peut faire à l'approche précédente est le grand coût en mémoire: un arbre comportant n feuilles sera représenté par une structure de données de taille proportionnelle à n. On utilise donc la mécanique des macro-constructeurs CEYX pour coder de façon minimale l'information.

Pour cela il existe une fonction de définition de modules paramétrés dans le langage HELL de nom "defmacroblock". Au premier abord la syntaxe est similaire à celle d'une définition de fonction:

```
(de arbre (n) ...)
```

s'écrit avec "defmacroblock" de la façon suivante:

```
(defmacroblock arbre (n) ...)
```

Une fonction "arbre" ayant un paramètre est alors définie, et l'appel peut se faire de la même façon par "(block (arbre <n>) ...)". Cependant la seule information gardée dans la structure de description du circuit sera "arbre" et "n", et non pas tout l'arbre. Les processeurs devront alors se charger de remplacer cet appel virtuel à une fonction, par le résultat de l'évaluation. Cette expansion, à la demande, se programmera tout naturellement en CEYX, et permet donc un codage minimal de l'information.

Defmacroblock permet aussi de typer les arguments, toujours grâce à la mécanique générale CEYX, et donc la meilleure écriture de "arbre" sera:

```
(defmacroblock arbre ( n ~ integer) ...)
```

Un dernier avantage subjectif de "defmacroblock" est de séparer le monde HELL du monde LISP, en permettant d'oublier complètement "de".

3.2.5 Mise en série de modules

De même que de nombreux problèmes algorithmiques sont intrinsèquement séquentiels, de nombreuses architectures de circuits sont formées de modules mis en série. C'est en particulier le cas dans la méthologie préconisée par Mead et Conway [Mead80], dans laquelle des parties combinatoires séquentielles sont mises en série avec des transistors de passage commandés par une horloge multi-phases.

Supposons, par exemple, que l'on veuille définir une porte composée d'un NAND, d'un NOT, et d'un transistor commandé par une horloge. Sans opérateur particulier, la porte pourra se décrire par:

```
(connect
     (wires
        in (entrée-1 entrée-2)
        in global (horloge)
        out (sortie)
        intern (sortie-du-NAND sortie-du-NOT))
     (block NAND  entrée-1 entrée-2 sortie-du-NAND)
     (block NOT   sortie-du-NAND sortie-du-NOT)
     (block trn   horloge sortie-du-NOT sortie))
```

On constate la lourdeur de l'écrire, surtout dûe à l'obligation de nommage par l'utilisateur des fils internes.

Avec le macro-constructeur "serial", on écrira simplement:

```
(serial
     (wires
        in (entrée-1 entrée-2)
```

```
        in global (horloge)
        out (sortie))
    (block NAND  entrée-1 entrée-2 +)
    (block NOT   - +)
    (block trn   horloge - sortie))
```

ou même:

```
(serial
    (wires
        in global (horloge))
    (block NAND  - +)
    (block NOT   - +)
    (block trn   horloge - +))
```

En effet à l'intérieur d'un "serial", les mots clefs "-" et "+" prennent le sens particulier d'entrée et de sortie. Les fils d'entrées d'un sous-nœud sont reliés aux fils de sortie du sous-nœud précédent, sauf dans le cas du premier sous-nœud, pour lequel les entrées deviennent entrées du nœud serial lui-même. De façon similaire, les sorties du dernier sous-nœud sont sorties du nœud serial. L'ordre des sous-nœuds est donc particulièrement important.

Les fils correspondant aux "-" et "+" sont ajoutés aux fils déjà déclarés et sont dans l'ordre entrées, sorties, fils donnés par l'utilisateur.

3.2.6 Réplications de modules

Une autre application intéressante de la mécanique des macro-constructeurs et de la structuration des fils est le macro-constructeur "iterconnect" et son pendant "iterserial" permettant de connecter plusieurs sous-nœuds éventuellement paramétrés. Cela procure un mécanisme simple d'itération, largement suffisant dans la plupart des cas.

La syntaxe a été empruntée à celle de la fonction lisp "for" qui permet une itération avec variable locale de boucle. La syntaxe de for est:

```
(for (<variable> <valeur_initiale> {<pas_(1_par_défaut)>} <valeur_finale>)
        <suite_d'instructions>)
```

et celles de iterconnect et iterserial:

```
(iterconnect
    (<variable> <valeur_initiale> {<pas_(1_par_défaut)>} <valeur_finale>)
    <declarations_comme_pour_connect>
    <noeud-hell_paramétré>)

(iterserial
    (<variable> <valeur_initiale> {<pas_(1_par_défaut)>} <valeur_finale>)
    <declarations_comme_pour_serial>
    <noeud-hell_paramétré>)
```

Ainsi pour juxtaposer N portes NOT et en faire une seule porte ayant un bus d'entrée et un bus de sortie, il suffit d'écrire:

```
(iterconnect (i 1 N)
             (wires in bus (entrée 1 N) out bus (sortie 1 N))
```

```
(block NOT [entrée i] [sortie i]))
```

et cela sera équivalent, pour N valant 3 à:

```
(connect
     (wires in bus (entrée 1 3) out bus (sortie 1 3))
     (block NOT [entrée 1] [sortie 1])
     (block NOT [entrée 2] [sortie 2])
     (block NOT [entrée 3] [sortie 3]))
```

L'utilisation de "iterserial" permet de décrire des structures relativement complexes. Soient par exemple 3 tranches identiques, numérotées de 0 à 2, instances d'un module "calcul", ayant 2 entrées Ain et Bin, 3 sorties Aout,Bout et Res, et que l'on veut mettre bout à bout de telle sorte que l'entrée Ain (resp. Bin) d'une tranche soit connectée à la sortie Aout (resp. Bout) de la tranche précédente. On veut en outre faire ressortir le Ain (resp. Bin) de la première tranche et le Aout (resp. Bout) de la dernière tranche, et faire traverser l'ensemble des tranches par un bus "Res" en indiçant les noms des fils (Res[0] Res[1] Res[2]). On suppose en outre que les paramètres de "calcul" sont dans l'ordre Ain, Aout, Res, Bin, Bout.

Pour faire tout cela, il suffira d'écrire:

```
(iterserial (i 0 2)
            (wires out bus (Res 0 2))
            (block calcul - + [Res i] - +))
```

En effet, cette expression est équivalente, après expansion, à:

```
(connect
   (wires
      in  (Ain Bin)
      out (Aout Bout)
      out bus (Res 0 2)
      intern (Aout[0] Bout[0] Aout[1] Bout[1]))
   (block calcul Ain     Aout[0] Res[0] Bin     Bout[0])
   (block calcul Aout[0] Aout[1] Res[1] Bout[0] Bout[1])
   (block calcul Aout[1] Aout    Res[2] Bout[1] Bout   ))
```

et s'utilisera par exemple dans l'expression:

```
(block  (iterserial ...)
        mon-Ain mon-Bin mon-Res mon-Aout mon-Bout)
```

où mon-Res est un fil structuré à 3 composantes.

On voit sur cet exemple la puissance qu'il est facile d'atteindre grâce au langage universel de programmation sous-jacent. Bien sûr, le code de "serial" est relativement complexe (plusieurs dizaines de lignes), mais on peut espérer qu'il résoud une classe suffisamment large de problèmes pour qu'on n'ait à l'écrire qu'une seule fois.

3.3 Syntaxe B.N.F. du langage

3.3.1 Convention

<liste_d'objets>::=(<objet1><objet2> ...<objetn>) n>=0
<suite_d'objets>::= <objet1><objet2> ...<objetn> n>=0

3.3.2 Micro-langage

```
<definition_de_symbole_HELL>::=
      (defblock
            <symbole_HELL>                          ; non évalué
            <suite_de_typages_de_fil>)
            <suite_de_noeuds_HELL_avec_io_nommes>)      ; évaluée

<noeud_HELL_avec_io_nommes>::=
      (block
            <symbole_HELL>                          ; évalué
            <suite_de_noms_de_paramètres>)              ; non évaluée

<typage_de_fil>::=
      <typage> <liste_de_noms_de_fil>                   ; non évaluée
        |
      <typage> eval <liste_de_noms_de_fil>          ; évaluée

<typage>::=
      <genre>
      |intern|interne|local

<genre>::=
      input|in|entree
      |
      output|out|sortie
      |
      io|inout|es

<nom_de_fil>::=
      <symbole>

<symbole_HELL>::=
      <symbole>
```

3.3.3 Macro-langage

```
<noeud_HELL>::=
      <objet_HELL> | <symbole_HELL>

<objet_HELL>::=
      (connect
            <déclarations>                          ; évalué
            <suite_de_noeuds_HELL_avec_io_nommes>)      ; évaluée
      |
      (lconnect
            <déclarations>                          ; évalué
            <liste_de_noeuds_HELL_avec_io_nommes>)      ; évaluée
      |
      (iterconnect
            <description_de_boucle>
            <déclarations>                          ; évalué
            <noeud_HELL_avec_io_nommes>)            ; évalué
      (serial
            <déclarations>                          ; évalué
            <suite_de_noeuds_HELL_avec_io_nommes>)      ; évaluée
      |
      (lserial
            <déclarations>                          ; évalué
            <liste_de_noeuds_HELL_avec_io_nommes>)      ; évaluée
      |
      (iterserial
            <description_de_boucle>
            <déclarations>                          ; évalué
            <noeud_HELL_avec_io_nommes>)            ; évalué


<noeud_HELL_avec_io_nommes>::=
      (block
            <noeud_HELL>                      ; évalué
            <suite_de_noms_de_paramètres>)            ; non évaluée

<déclarations>::=
      (wires
            <suite_de_typages_de_fil>)

<typage_de_fil>::=
      <typage> <liste_de_noms_de_fil>                 ; non évaluée
      |
      <typage> eval <liste_de_noms_de_fil>        ; évaluée
      |
      <typage> bus (<nom_de_fil> <min> <max>)         ; non évalué
      |
      <typage> list (<nom_de_fil> <liste_de_noms_de_paramètre>)

<typage>::=
      <genre>
```

```
        |<genre> global
      |intern|interne|local

<genre>::=
      input|in|entree
      |
      output|out|sortie
      |
      io|inout|es

<definition_de_symbole_HELL>::=
      (defblock
            <symbole_HELL>                              ; non évalué
            <déclarations>                              ; évalué
            <suite_de_nœuds_HELL_avec_io_nommes>)       ; évaluée

<nom_de_paramètre>::=
      <nom_de_fil>
      |
      <liste_de_noms_de_paramètres>

<nom_de_fil>::=
      <symbole>                             ; non évalué
      |
      %<expression_retournant_un_symbole>         ; évaluée

<symbole_HELL>::=
      <symbole>

<description_de_boucle>::=
      (<symbole> <valeur_initiale> <valeur_finale>)
      |
      (<symbole> <valeur_initiale> <incrément> <valeur_finale>)
```

3.4 Problèmes syntaxiques

L'aspect syntaxique d'autres langages de description électrique est souvent celui d'un langage de programmation existant comme Pascal (Ilisdl [Ilisdl82] par exemple). Cela évite la multiplicité des parenthèses, et donc augmente, pour le néophyte LISP, la facilité de lecture. L'utilisation du "%" pour évaluer les paramètres est à la limite du tolérable et a son origine dans le problème omniprésent de l'évaluation en LISP. Il faudra sans doute donner à HELL un aspect syntaxique plus plaisant, mais cela n'est pas si facile, car il faut pouvoir garder toute la puissance sous jacente de LISP, et mélanger des fonctions LISP simples à une description HELL.

Un autre problème syntaxique, mais qui ne doit rien à LISP provient de la stratégie de nommage des ports d'entrée sortie d'un module. En effet, pour raccorder la sortie d'un NOT à la grille d'un transistor, il peut être plus lisible d'écrire quelque chose comme "NOT.sortie = transistor.grille" que de donner un nom explicite au signal. Cela évite alors ainsi la déclaration des signaux

internes. La pratique semble cependant montrer que, dans la plupart des cas, ces déclarations ne sont guères contraignantes. On notera que la sémantique d'un module doit alors être légèrement différente: sont visibles de l'extérieur d'un module non seulement le typage (sens et structure) des ports d'entrée sortie, mais aussi leurs noms.

Cela permet alors de donner la liste des paramètres d'une instance de module de deux façons: de façon positionnelle, comme c'est actuellement le cas en HELL, ou en précisant quel paramètre est à mettre en relation avec quel nom de la définition. C'est ce choix qui a été fait en Ada [Ada83] (ou en Mesa [Mesa82]), où il est possible d'appeler une fonction "trans" définie par:

```
function trans (in grille : signal ; inout source , drain : signal)
```

de deux façons différentes:

```
trans (x , y , z) ,
```

mais aussi:

```
trans (x , drain => z , source => y) .
```

Dans certains langages de description de circuits les deux mécanismes sont donnés [Hisdl82], et il faudra un jour faire de même en HELL .

4 Description de circuits en HELL

4.1 Introduction

Dans cette section, nous allons donner un bref aperçu des processeurs autres que ceux de simulation, afin de montrer comment la chaîne de CAO s'articule autour du langage HELL.

4.2 Facilités d'entrée

Pour les communications homme-machine, il n'y a que deux dispositifs couramment utilisés: l'écran alphanumérique associé à un clavier, et l'écran graphique associé à un dispositif de pointage.
En dehors de tout problème de coûts, tant matériels que logiciels, il apparaît que l'entrée textuelle d'informations présente de nombreux avantages: concision, décompilation triviale, précision, matériel standard. En particulier, il semble très difficile d'entrer graphiquement des données relatives à un paramétrage. Il est plus facile d'écrire un if .. then .. else .. que de le dessiner.

La forme d'entrée la mieux appropriée pour décrire structurellement des circuits est donc textuelle. Le concepteur va donc écrire le code HELL sous

éditeur.
Comme la frappe des mots clefs du langage et le parenthésage des expressions sont des opérations lentes, l'éditeur va fournir des facilités d'abréviations. Ces mêmes facilités vont aussi servir à exécuter par simple manipulation de quelques touches, des opérations plus complexes que la simple frappe du texte, comme des déplacements ou modifications structurelles [Hullot84, Mentor75].

Mais le rôle le plus important d'un éditeur n'est pas la simple création de texte, mais plutôt la modification d'une description. Là encore, l'éditeur doit fournir le moyen à l'utilisateur de se déplacer rapidement de façon structurelle ou non, de visualiser sur plusieurs fenêtres des morceaux différents de la structure, eventuellement même d'adopter la syntaxe que l'utilisateur préfère.

4.3 Facilités de visualisation

Si l'introduction textuelle présente de nombreux avantages par rapport à l'entrée graphique, il n'en est pas de même pour la visualisation et la relecture du code. Le concepteur doit avoir la possibilité de documenter le code HELL par des schémas, plus ou moins éloignés de la structure géométrique qu'aura le circuit une fois décrit jusqu'au niveau du dessin des masques.

Les algorithmes et heuristiques de décompilation schématique de morceaux de circuits décrits en HELL restent à établir. Le problème le plus ardu semble être la représentation de l'itération, par exemple de circuits décrits avec "slices". Certains langages présentent des solutions, mais qui ne semblent guère satisfaisantes (par exemple [Alex80]), car elles manquent de généralité et sont d'une relecture difficile.

4.4 Liens avec la géometrie

A chaque module défini par defblock, il est possible d'associer une sémantique graphique, par exemple dans le formalisme LUCIFER. On peut aussi donner une sémantique aux macro-constructeurs (définis par defmacroblock), et ainsi chaque macro-constructeur devient un morceau de compilateur de silicium. Par exemple, on peut donner à un macroconstructeur d'arbre une sémantique qui décompilera l'arbre de multiples façons: arbre en H, arbre linéaire, arbre en T, arbre rectangulaire, etc ...
Le choix de la décompilation doit, dans un premier temps, être laïssé au concepteur. On peut cependant espérer que dans quelques années, on soit capable d'écrire un mini-système expert capable de faire de façon heuristique un tel choix. Le compilateur de silicium apparaîtra donc comme un ensemble de procédures, de sémantiques, et d'heuristiques permettant de raffiner la structure électrique telle qu'elle est définie en HELL, en y ajoutant des informations suffisantes pour permettre la génération de géométrie.

4.5 Conclusion

Pour avoir unicité de la description, certains langages de description de circuits multiplient les informations sémantiques à l'intérieur de la description structurelle, et comme le processeur le plus facilement disponible est le simulateur, de nombreux langages mélangent description structurelle et description fonctionnelle. Parfois aussi des indications géométriques sont mélangées au texte même [Zeus83].

Cependant, l'approche classique consiste à faire coexister plusieurs descriptions, et à établir des liens entre ces structures. Beaucoup d'hommes-années sont ainsi perdues à écrire des interfaces, et à traduire des descriptions d'un formalisme à l'autre, chaque processeur ayant ses propres langages d'entrée et de sortie. En outre, certaines possibilités, comme l'édition simultanée de toutes les structures (puisqu'il n'y en a qu'une !) sont impossibles.

Pour HELL, la mécanique des sémantiques de CEYX permet de conserver tous les avantages d'une structure unique, tout en rajoutant n'importe où dans la structure des informations complémentaires relatives aux différents processeurs.

5 Conclusion sur le langage HELL

La description structurelle de modules communiquant entre eux est un type de description tout à fait général, et HELL devrait permettre de décrire aussi bien des circuits intégrés, des circuits imprimés, des machines, des réseaux locaux, ou même de grands réseaux. Dans tous ces cas, l'ensemble des mécanismes introduits par HELL pour les VLSI semble nécessaire mais suffisant.

Il semble donc qu'il ne faille pas fonder un système de CAO de circuits sur un langage de description de masques, mais sur un langage bien plus général de description de structures, tout en prenant soin de ne pas rajouter dans la définition du langage des informations particulières à un processeur.
Cela permet en effet d'éviter une différenciation entre les objets terminaux de la structure (feuilles) et les non-terminaux (nœuds), puisque tout module peut être considéré comme feuille pour un processeur, et comme nœud pour un autre, et rend donc le système incrémental: il est possible d'écrire de nouveaux processeurs, et de rajouter de la sémantique sur les primitives qu'ils reconnaissent, sans pour autant modifier quoi que ce soit dans le langage ou dans les descriptions dans ce langage.

Ainsi, s'il manque pour l'instant de nombreux processeurs utilisant la structure HELL, il semble que ce langage purement structurel soit bien adapté à l'écriture de tels processeurs.

BIBLIOGRAPHIE

[Ada83] : Reference Manual for the Ada programming language, Ansi/Mil Standard, Janvier 83.

[Alex80] : Alex, a conversational, hierarchical logic design system, K. A. Duke, 17th DAC, 1980.

[Cascade83] : Hierarchical mixed-mode simulation mechanisms in the cascade project, D. Borrione, M. Humbert, C. Le Faou, VLSI83 proceedings, Aout 83.

[Chailloux80] : Le modele Vlisp : description, évaluation et interprétation, thèse de 3eme cycle, universite de Paris 6, Avril 80.

[Chailloux82] : LE_LISP 68k, Le Manuel Rapport INRIA, Juillet 82.

[Gallot83] : Vérification des règles de dessin, L. Gallot, thèse de 3ème cycle, Octobre 83.

[Greenberg79] : Emacs Text Editor User's Guide, B. S. Greenberg, Honeywell Information Systems Inc., 1979.

[Greussay77] : Contribution à la définition interprétative et à l'implémentation des lambda-langages, thèse, Université de Paris 6, Novembre 77.

[Heintz82] : Un extracteur de circuits intégrés, C. Heintz, thèse de 3ème cycle, Octobre 82.

[Hisdl82] : HISDL: a structure description language, par Lim, communication ACM novembre 82 .

[Hullot83] : CEYX, a Multiformalism Programming Environment, J.-M. Hullot, IFIP 83, Septembre 83.

[Hullot84] : BIGMACS, a structured editor. J.-M. Hullot, Rapport Inria à paraître.

[Lévy83] : Manuel d'utilisation de l'éditeur graphique LUCIOLE, J.-J. Lévy, Avril 83.

[LUCIFER83] : Le système LUCIFER d'aide à la conception, J. Chailloux, J.-M. Hullot,J.-J. Lévy et J. Vuillemin, Mars 83.

[McCarthy62] : LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass.

[Mesa82] : Manuel de l'utilisateur Mesa, publication interne Xerox.

[Moto-oka83] : The role of Vlsi Cad in the fifth generation computer project, Tohru Moto-Oka, University of Tokyo.

[Smalltalk82] : Smalltalk-80 : the language and its implementation, Goldberg et Robson, Addison Wesley Publishing Company.

Table des matières

1 Nécessité d'un langage de description de circuits intégrés 3
2 Le langage de programmation 'CEYX' 3
2.1 Le langage d'implémentation LE_LISP 4
2.1.1 Historique 4
2.1.2 Structures de contrôle 4
2.1.3 Implémentation et portabilité 5
2.1.4 Structures de données 5
2.2 Types 5
2.2.1 Types simples 6
2.2.2 Les types composés list et array 6
2.2.3 Le type composé record 6
2.2.4 Association de noms aux types 6
2.2.5 Typage des objets 7
2.3 Messages et sémantiques 7
2.3.1 Exécution d'une sémantique 7
2.3.2 Surcharge d'opérateurs 8
2.3.3 Dépendance hiérarchique des sémantiques 8
2.3.4 Sémantiques ou procédures ? 8
2.4 Le type arbre 9
2.4.1 Description 9
2.4.2 Univers 9
2.4.3 Constructeurs 10
2.4.4 Macro-constructeurs 10
2.5 Conclusion 10
3 Description du langage HELL 11
3.1 Description du micro-langage 11
3.1.1 En tête d'un module 11
3.1.2 Connection de plusieurs modules 12
3.1.3 Fils locaux à un module 13
3.1.4 Définition de module 13
3.1.5 Instance d'un module 14
3.1.6 Typage des ports d'entrée sortie 14
3.1.7 Modules primitifs 14
3.2 Description du macro-langage 15
3.2.1 Paramètres implicites 15
3.2.2 Structuration des fils 16
3.2.3 Immersion dans un langage de programmation 17
3.2.4 Macro-constructeurs de modules 18
3.2.5 Mise en série de modules 19
3.2.6 Réplications de modules 20
3.3 Syntaxe B.N.F. du langage 22
3.3.1 Convention 22
3.3.2 Micro-langage 22
3.3.3 Macro-langage 23
3.4 Problèmes syntaxiques 24
4 Description de circuits en HELL 25
4.1 Introduction 25
4.2 Facilités d'entrée 25
4.3 Facilités de visualisation 26
4.4 Liens avec la géometrie 26
4.5 Conclusion 27

Table des matières

7 Conclusion sur le langage HELL 27

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique